```
location ~ \.php$ {
  set $args "${query_string}&group=${variable}";
}
```

SSL and security

Nginx provides secure HTTP functionalities through the SSL module, but also offers an extra module called *Secure Link* that helps you protect your website and visitors in a totally different way.

SSL

The SSL module enables HTTPS support, HTTP over SSL/TLS in particular. It gives you the option to serve secure websites by providing a certificate, a certificate key, and other parameters defined with the following directives:

This module is not included in the default Nginx build.

Directive	Description
`ssl` Context: `http`, `server`	Enables HTTPS for the specified server. This directive is the equivalent of `listen 443 ssl` or `listen port ssl` more generally. Syntax: `on` or `off` Default: `ssl off;`
`ssl_certificate` Context: `http`, `server`	Sets the path of the PEM certificate. Syntax: File path
`ssl_certificate_key` Context: `http`, `server`	Sets the path of the PEM secret key file. Syntax: File path
`ssl_client_certificate` Context: `http`, `server`	Sets the path of the client PEM certificate. Syntax: File path
`ssl_crl` Context: `http`, `server`	Orders Nginx to load a CRL (Certificate Revocation List) file, which allows checking the revocation status of certificates.
`ssl_dhparam` Context: `http`, `server`	Sets the path of the *Diffie-Hellman* parameters file. Syntax: File path.

Directive	**Description**
`ssl_protocols` Context: `http`, `server`	Specifies the protocol that should be employed. Syntax: `ssl_protocols [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2];` Default: `ssl_protocols TLSv1 TLSv1.1 TLSv1.2;`
`ssl_ciphers` Context: `http`, `server`	Specifies the ciphers that should be employed. The list of available ciphers can be obtained by running the following command from the shell: `openssl ciphers`. Syntax: `ssl_ciphers cipher1[:cipher2...];` Default: `ssl_ciphers ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;`
`ssl_prefer_server_ciphers` Context: `http`, `server`	Specifies whether server ciphers should be preferred over client ciphers. Syntax: `on` or `off` Default: `off`
`ssl_verify_client` Context: `http`, `server`	Enables verifying certificates to be transmitted by the client and sets the result in the `$ssl_client_verify`. The `optional_no_ca` value verifies the certificate if there is one, but does not require it to be signed by a trusted CA certificate. Syntax: `on` | `off` | `optional` | `optional_no_ca` Default: `off`
`ssl_verify_depth` Context: `http`, `server`	Specifies the verification depth of the client certificate chain. Syntax: Numeric value Default: `1`
`ssl_session_cache` Context: `http`, `server`	Configures the cache for SSL sessions. Syntax: `off`, `none`, `builtin:size` or `shared:name:size` Default: `off` (disables SSL sessions)
`ssl_session_timeout` Context: `http`, `server`	When the SSL sessions are enabled, this directive defines the timeout for using session data. Syntax: Time value Default: 5 minutes

Directive	**Description**
`ssl_password_phrase` Context: `http`, `server`	Specifies a file containing the passphrases for secret keys. Each passphrase is specified on a separate line; they are tried one after the other when loading a certificate key. Syntax: file name Default: (none)
`ssl_buffer_size` Context: `http`, `server`	Specifies the buffer size when serving requests over SSL. Syntax: Size value Default: 16k
`ssl_session_tickets` Context: `http`, `server`	Enables TLS session tickets, allowing the client to reconnect faster by skipping re-negotiation. Syntax: `on` or `off` Default: `on`
`ssl_session_ticket_key` Context: `http`, `server`	Sets the path of the key file used to encrypt and decrypt the TLS session tickets. By default, a random value is generated. Syntax: file name Default: (none)
`ssl_trusted_certificate` Context: `http`, `server`	Sets the path of a trusted certificate file (PEM format) used to validate the authenticity of client certificates as well as the stapling of OCSP responses. More about SSL stapling can be found later on in the chapter. Syntax: file name Default: (none)

Additionally, the following variables are made available:

- `$ssl_cipher`: Indicates the cipher used for the current request
- `$ssl_client_serial`: Indicates the serial number of the client certificate
- `$ssl_client_s_dn` and `$ssl_client_i_dn`: Indicates the value of the Subject and Issuer DN of the client certificate
- `$ssl_protocol`: Indicates the protocol in use for the current request
- `$ssl_client_cert` and `$ssl_client_raw_cert`: Returns the client certificate data, which is raw data for the second variable
- `$ssl_client_verify`: Set to `SUCCESS` if the client certificate was successfully verified
- `$ssl_session_id`: Allows you to retrieve the ID of an SSL session

Setting up an SSL certificate

Although the SSL module offers a lot of possibilities, in most cases only a couple of directives are actually useful for setting up a secure website. This guide will help you to configure Nginx to use an SSL certificate for your website (in the example, your website is identified by `secure.website.com`). Before doing so, ensure that you already have the following elements at your disposal:

- A `.key` file generated with the following command: `openssl genrsa -out secure.website.com.key 1024` (other encryption levels work too).
- A `.csr` file generated with the following command: `openssl req -new -key secure.website.com.key -out secure.website.com.csr`.
- Your website certificate file, as issued by the Certificate Authority, for example, `secure.website.com.crt`. (Note: In order to obtain a certificate from the CA, you will need to provide your `.csr` file.)
- The CA certificate file as issued by the CA (for example, `gd_bundle.crt` if you purchased your certificate from `http://www.GoDaddy.com`).

The first step is to merge your website certificate and the CA certificate together with the following command:

```
cat secure.website.com.crt gd_bundle.crt > combined.crt
```

You are then ready to configure Nginx for serving secure content:

```
server {
    listen 443;
    server_name secure.website.com;
    ssl on;
    ssl_certificate /path/to/combined.crt;
    ssl_certificate_key /path/to/secure.website.com.key;
    […]
}
```

SSL Stapling

SSL Stapling, also called **Online Certificate Status Protocol** (**OCSP**) Stapling, is a technique that allows clients to easily connect and resume sessions to an SSL/TLS server without having to contact the Certificate Authority, thus reducing the SSL negotiation time. In normal OCSP transactions, the client normally contacts the Certificate Authority so as to check the revocation status of the server's certificate. In the case of high traffic websites, this can cause a huge stress on the CA servers. An intermediary solution was designed—Stapling. The OCSP record is obtained periodically from the CA by your server itself, and is *stapled* to exchanges with the client. The OCSP record is cached by your server for a period of up to 48 hours in order to limit communications with the CA.

Enabling SSL Stapling should thus speed up the communication between your visitors and your server. Achieving this in Nginx is relatively simple: all you really need is to insert three directives in your server block, and obtain a full trusted certificate chain file (containing both the root and intermediate certificates) from your CA.

- `ssl_stapling on`: enables SSL Stapling within the server block
- `ssl_stapling_verify on`: enables verification of OCSP responses by the server
- `ssl_trusted_certificate filename`: where `filename` is the path of your full trusted certificate file (extension should be .pem).

Two optional directives also exist, which allow you to modify the behavior of this module:

- `ssl_stapling_file filename`: where `filename` is the path of a cached OCSP record, overriding the record provided by the OCSP responder specified in the certificate file.
- `ssl_stapling_responder url`: where `url` is the URL of your CA's OCSP responder, overriding the URL specified in the certificate file.

If you are having issues connecting to the OCSP responder, make sure your Nginx configuration contains a valid DNS resolver (using the `resolver` directive).

SPDY

The SPDY module offers support for the SPDY protocol (the SPDY module is not included by default). You can enable SPDY on your server by appending the keyword `spdy` at the end of your `listen` directive.

```
server {
    listen 443 ssl spdy;
    [...]
}
```

Due to the nature of SPDY, it can only be enabled over SSL. Two directives and two variables are brought in by this module:

- `spdy_chunk_size`: sets the size of the SPDY chunks
- `spdy_headers_comp`: sets the compression level for headers (0 to disable, 1 to 9 from lowest/fastest to highest/slowest compression)
- `$spdy`: this variable contains the SPDY protocol version if SPDY is used, an empty string otherwise
- `$spdy_request_priority`: this variable indicates the request priority if SPDY is used, an empty string otherwise

SPDY is a protocol developed by Google, aiming to improve web latency and security. Although its utility was demonstrated (albeit not always significantly), Google decided to abandon the project after the HTTP/2 standard was ratified. As a result, SPDY support will be officially withdrawn during the first half of 2016.

Secure link

Totally independent from the SSL module, Secure link provides basic protection by checking the presence of a specific hash in the URL before allowing the user to access a resource:

```
location /downloads/ {
secure_link_md5 "secret";
secure_link $arg_hash,$arg_expires;
    if ($secure_link = "") {
      return 403;
    }
}
```